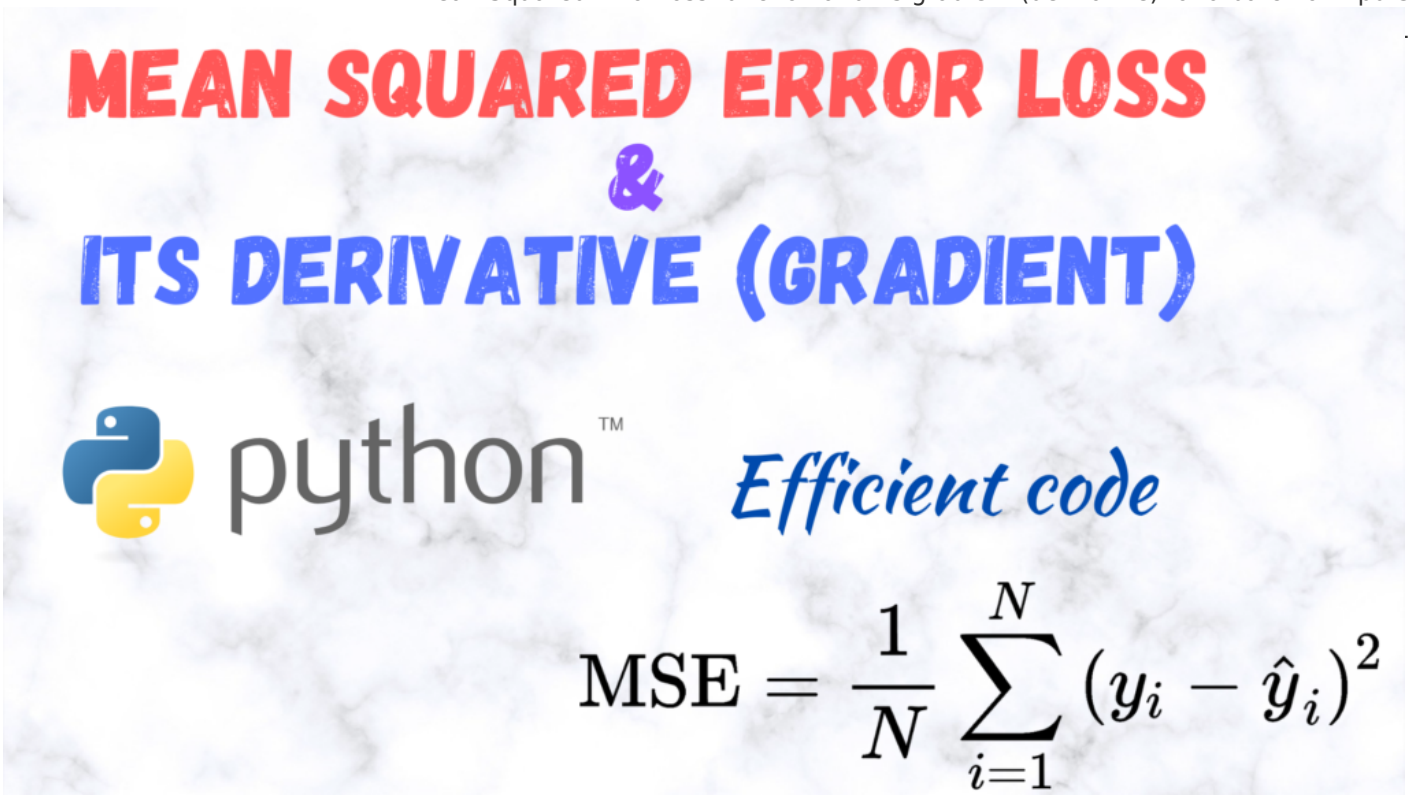
Mean Squared Error loss function and its gradient (derivative) for a batch of inputs



In this post, I show you how to implement the Mean Squared Error (MSE) Loss/Cost function as well as its derivative for Neural networks in Python. The function is meant for working with a batch of inputs, i.e., a batch of samples is provided at once.

The mathematical definition of the MSE loss function is

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

where $\hat{y}_i$ are the expected or target outputs (known beforehand), y_i are the predicted outputs from the neural network, and N is the number of samples.

The derivative of the MSE loss function is:

$$\text{MSE} = \frac{2}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)$$

Please note, that in the above definitions we are only considering a single output node. That is there is one output per sample.
But it is very common for neural networks to have more than one output node (multiple outputs) for a given input. In that case one has to average the squared error not only over the samples but also over the number of output nodes.

In this post I will only focus on this general case which means we would expect the predicted and target outputs to be 2D arrays, with nRows=nSamples, and nColumns=nOutput_Nodes. In other words, the rows correspond to outputs for different set of input samples, and the columns correspond to the outputs from the different output nodes.

Based on this we can implement the MSE loss function in Python as:

MSE loss simplest implementation

```
def MSE_loss(predictions, targets):
    """
    Computes Mean Squared error/loss between targets
    and predictions.
    Input: predictions (N, k) ndarray (N: no. of samples, k: no. of output nodes)
          targets (N, k) ndarray        (N: no. of samples, k: no. of output nodes)
    Returns: scalar
    Note: The averaging is only done over the output nodes and not over the samples in
a batch.
    Therefore, to get an answer similar to PyTorch, one must divide the result by the
batch size.
    """
    return np.sum((predictions-targets**2)/predictions.shape[1]
```

MSE loss derivative simplest implementation

```
def MSE_loss_grad(predictions, targets):
    """
    Computes mean squared error gradient between targets
    and predictions.
    Input: predictions (N, k) ndarray (N: no. of samples, k: no. of output nodes)
          targets (N, k) ndarray        (N: no. of samples, k: no. of output nodes)
    Returns: (N,k) ndarray
    Note: The averaging is only done over the output nodes and not over the samples in
a batch.
    Therefore, to get an answer similar to PyTorch, one must divide the result by the
batch size.
    """
    return 2*(predictions-targets)/predictions.shape[1]
```

The above functions return the MSE loss and its derivative respectively, only averaged over the output nodes. Averaging over the samples can then be done by simply dividing the result by nSamples as shown

```
average_MSE = MSE_loss(predictions, target)/predictions.shape[0]
average_MSE_grad = MSE_loss_grad(predictions, target)/predictions.shape[0]
```

These implementations can be further accelerated (sped-up) by using Numba (https://numba.pydata.org/). Numba is a Just-in-time (JIT) compiler that

> translates a subset of Python and NumPy code into fast machine code.

To use numba, install it as:

```
pip install numba
```

Also, make sure that your numpy is compatible with Numba or not, although usually pip takes care of that. You can

Accelerating the above functions using Numba is quite simple. Just modify them in the following manner:

MSE loss NUMBA implementation

```
from numba import njit
@njit(cache=True,fastmath=True)
def MSE_loss(predictions, targets):
    """
    Computes Mean Squared error/loss between targets
    and predictions.
    Input: predictions (N, k) ndarray (N: no. of samples, k: no. of output nodes)
          targets (N, k) ndarray        (N: no. of samples, k: no. of output nodes)
    Returns: scalar
    Note: The averaging is only done over the output nodes and not over the samples in
a batch.
    Therefore, to get an answer similar to PyTorch, one must divide the result by the
batch size.
    """
    return np.sum((predictions-targets)**2)/predictions.shape[1]
```

NOTE: I noticed that the above implementation can be unstable and return erroneous results for very large arrays. Therefore, you can use the following even faster implementation with parallelization:

```
from numba import njit
@njit(cache=True,fastmath=False, parallel=True)
def MSE_loss(predictions, targets):
    """
    Computes Mean Squared error/loss between targets
    and predictions.
    Input: predictions (N, k) ndarray (N: no. of samples, k: no. of output nodes)
          targets (N, k) ndarray        (N: no. of samples, k: no. of output nodes)
    Returns: scalar
    Note: The averaging is only done over the output nodes and not over the samples in
a batch.
    Therefore, to get an answer similar to PyTorch, one must divide the result by the
batch size.
    """
    loss = 0.0
    for i in prange(predictions.shape[0]):
        for j in range(predictions.shape[1]):
            loss = loss + (predictions[i,j] - targets[i,j])**2
    # Average over number of output nodes
    loss = loss / predictions.shape[1]
    return loss
```

MSE loss derivative NUMBA implementation

```
from numba import njit
@njit(cache=True,fastmath=True)
def MSE_loss_grad(predictions, targets):
    """
    Computes mean squared error gradient between targets
    and predictions.
    Input: predictions (N, k) ndarray (N: no. of samples, k: no. of output nodes)
          targets (N, k) ndarray        (N: no. of samples, k: no. of output nodes)
    Returns: (N,k) ndarray
    Note: The averaging is only done over the output nodes and not over the samples in
a batch.
    Therefore, to get an answer similar to PyTorch, one must divide the result by the
batch size.
    """
    return 2*(predictions-targets)/predictions.shape[1]
```

This is quite fast and competitive with Tensorflow and PyTorch.

It is in fact also used in the CrysX-Neural Network library (crysx_nn)

Furthermore, the above implementations can also be accelerated using Cupy (CUDA).

> CuPy is an open-source array library for GPU-accelerated computing with Python. CuPy utilizes CUDA Toolkit libraries to make full use of the GPU architecture.

The Cupy implementations look as follows:

```
def MSE_loss_cupy(predictions, targets):
    """
    Computes Mean Squared error/loss between targets
    and predictions.
    Input: predictions (N, k) ndarray (N: no. of samples, k: no. of output nodes)
          targets (N, k) ndarray        (N: no. of samples, k: no. of output nodes)
    Returns: scalar
    Note: The averaging is only done over the output nodes and not over the samples in
a batch.
    Therefore, to get an answer similar to PyTorch, one must divide the result by the
batch size.
    """
    return cp.sum((predictions-targets)**2)/predictions.shape[1]
```

```
def MSE_loss_grad_cupy(predictions, targets):
    """
    Computes mean squared error gradient between targets
    and predictions.
    Input: predictions (N, k) ndarray (N: no. of samples, k: no. of output nodes)
          targets (N, k) ndarray        (N: no. of samples, k: no. of output nodes)
    Returns: (N,k) ndarray
    Note: The averaging is only done over the output nodes and not over the samples in
a batch.
    Therefore, to get an answer similar to PyTorch, one must divide the result by the
batch size.
    """
    return 2*(predictions-targets)/predictions.shape[1]
```

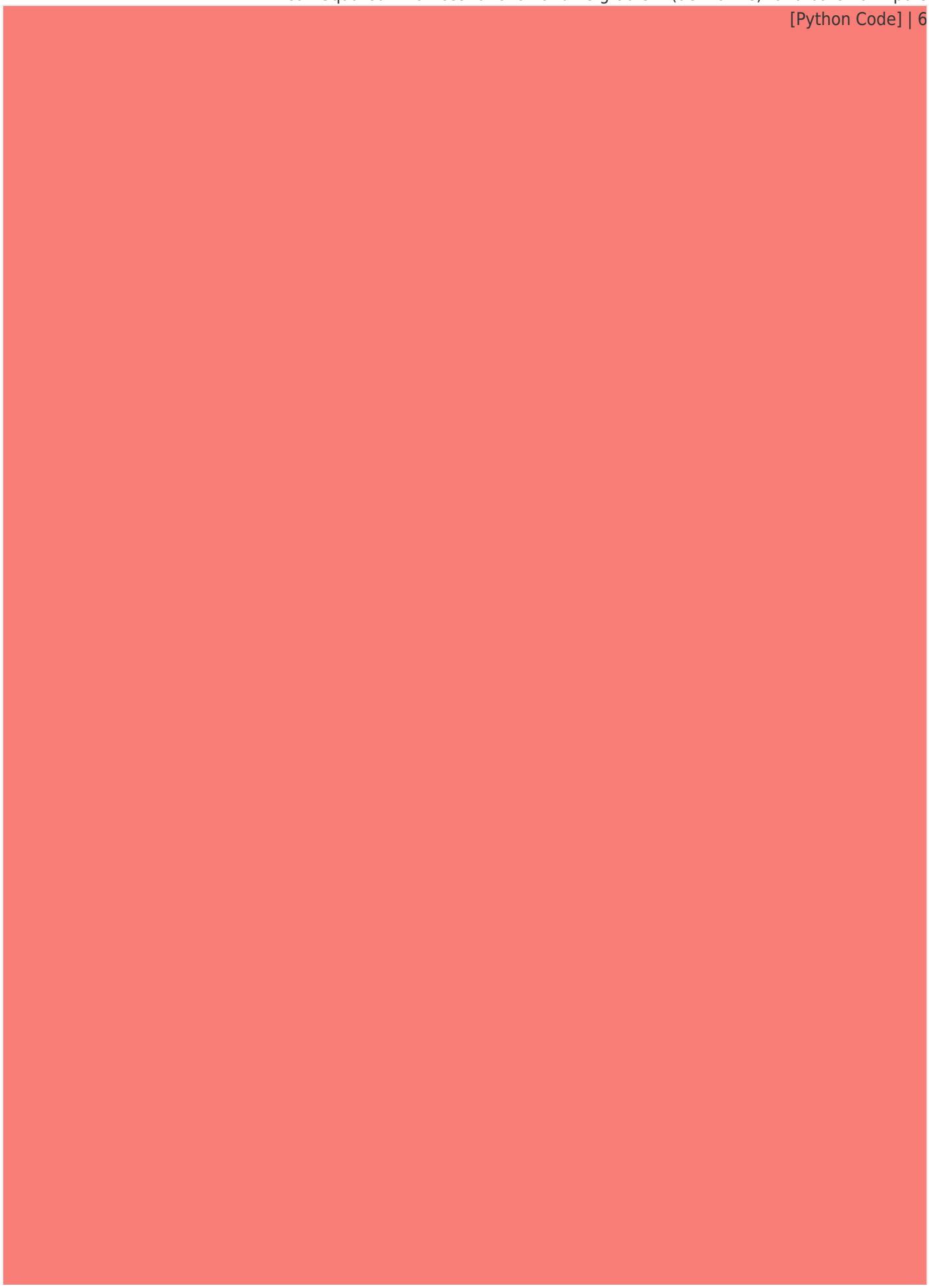
The above code is also used in the crysx_nn library.
I hope you found this information useful.

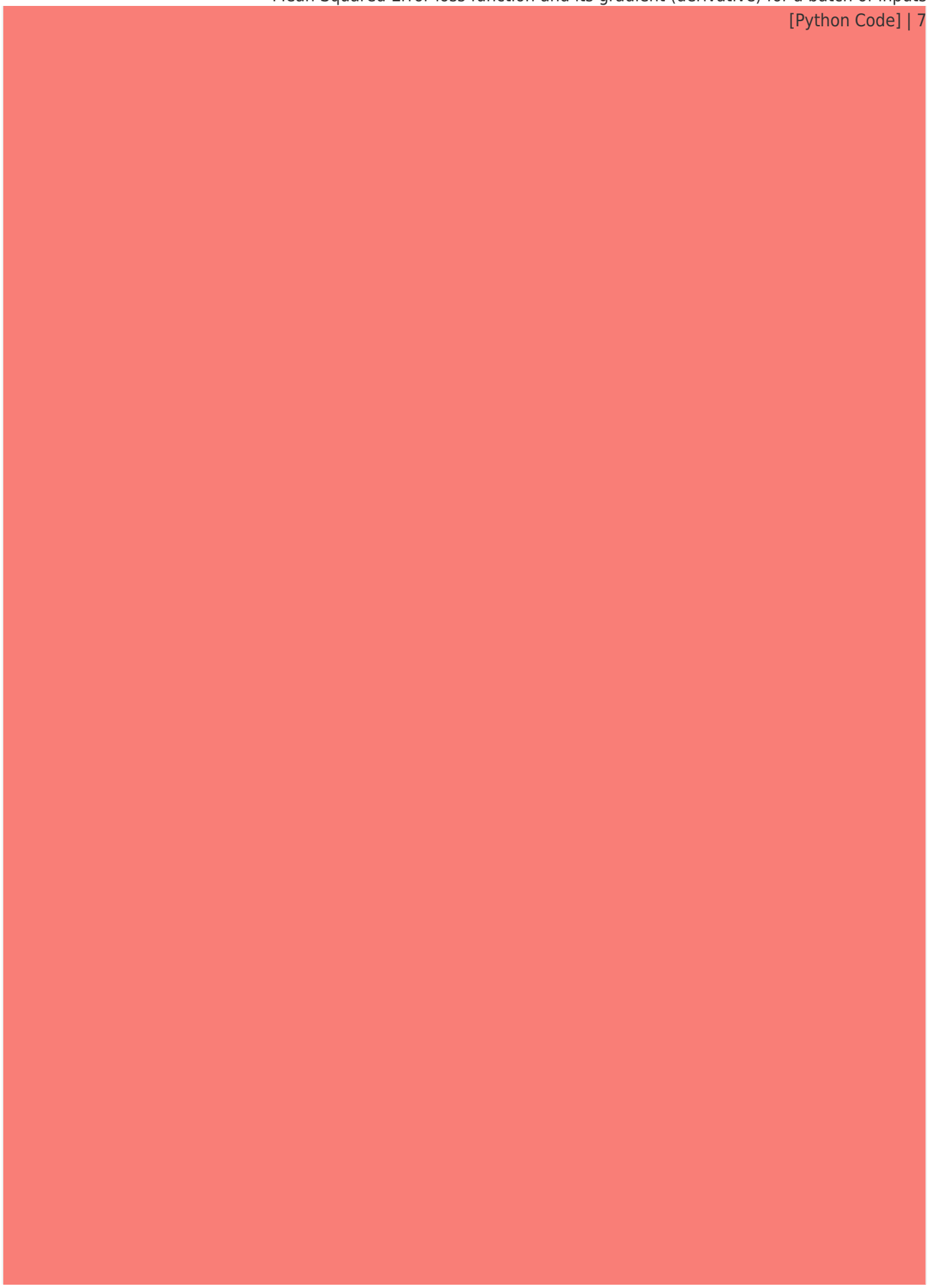
If you did, then don't forget to check out my other posts on Machine Learning and efficient implementations of activation/loss functions in Python.

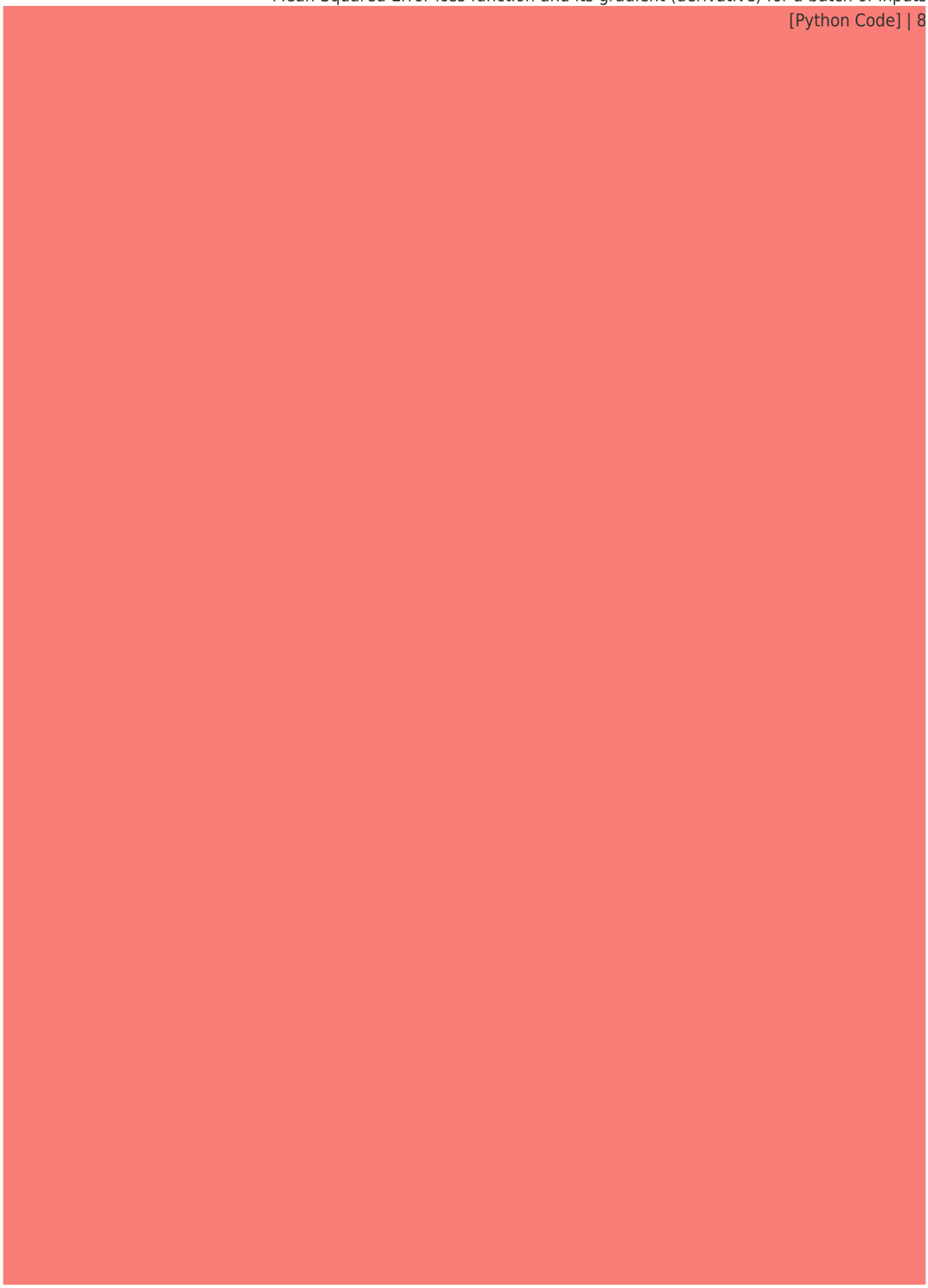
Manas Sharma

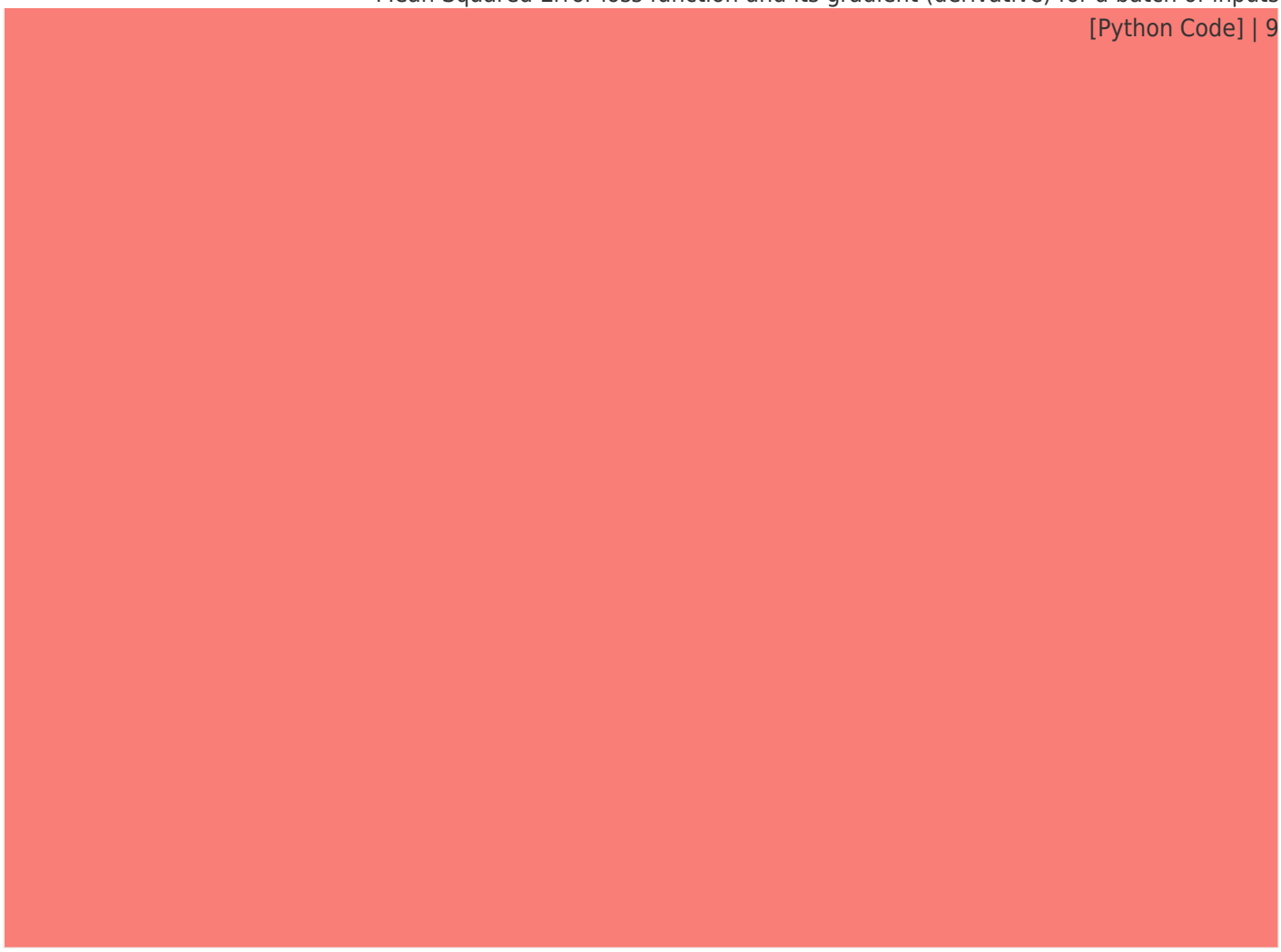
I'm a physicist specializing in computational material science with a PhD in Physics from Friedrich-Schiller University Jena, Germany. I write efficient codes for simulating light-matter interactions at atomic scales. I like to develop Physics, DFT, and Machine Learning related apps and software from time to time. Can code in most of the popular languages. I like to share my knowledge in Physics and applications using this Blog and a YouTube channel.

manas.bragitoff.com/









Share this:

Click to share on Facebook (Opens in new window)

Click to share on Twitter (Opens in new window)

Click to share on WhatsApp (Opens in new window)

Click to share on Pinterest (Opens in new window)

Click to share on Reddit (Opens in new window)

Click to share on LinkedIn (Opens in new window)

Click to email a link to a friend (Opens in new window)

Click to print (Opens in new window)

Click to share on Tumblr (Opens in new window)

Click to share on Pocket (Opens in new window)

Click to share on Telegram (Opens in new window)

[wpedon id="7041" align="center"]